图文 106 Broker是如何发送定时心跳的，以及如何进行故障感知?

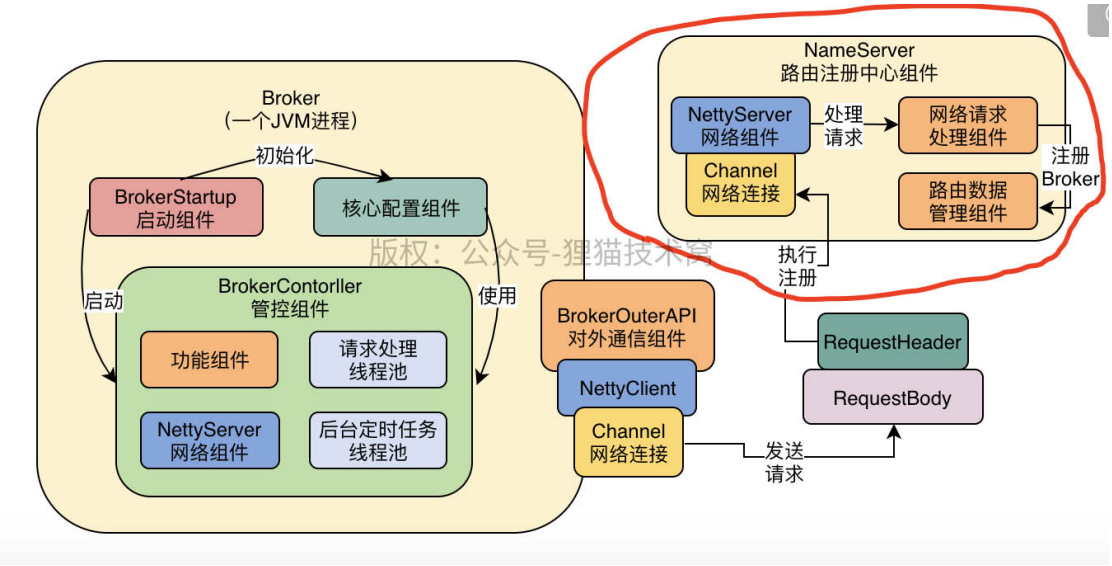
详情 评论

Broker是如何发送定时心跳的，以及如何进行故障感知?

昨天我们已经讲解了NameServer处理Broker注册请求的源码流程，大家已经知道了，NameServer核心其实就是基于Netty服务器来接收Broker注册请求，然后交给DefaultRequestProcessor这个请求处理组件，来处理Broker注册请求。

而真正的Broker注册的逻辑是放在RouteInfoManager这个路由数据管理组件里来进行实现的，最终Broker路由数据都会存放在RouteInfoManager内部的一些Map数据结构组成的路由数据表中。

我们看下图，就是一个示意。



今天我们就来讲讲，Broker是如何定时发送心跳到NameServer，让NameServer感知到Broker一直都存活着，然后如果Broker一段时间没有发送心跳到NameServer，那么NameServer是如何感知到Broker已经挂掉了。

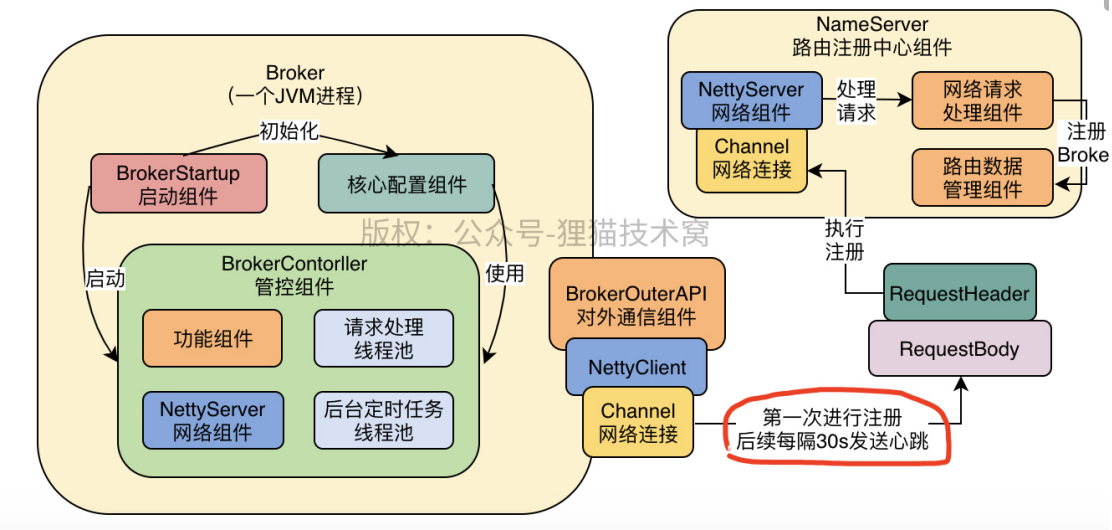
首先我们看一下Broker中的发送注册请求给NameServer的一个源码入口，其实就是在BrokerController.start()方法中，在BrokerController启动的时候，他其实并不是仅仅发送一次注册请求，而是启动了一个定时任务，会每隔一段时间就发送一次注册请求。

```
this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {

    @Override
    public void run() {
      try {
         BrokerController.this.registerBrokerAll(true, false, brokerConfig.isForceRegister());
      } catch (Throwable e) {
         log.error("registerBrokerAll Exception", e);
      }
    }
},
1000 * 10,
Math.max(10000, Math.min(brokerConfig.getRegisterNameServerPeriod(), 60000)), TimeUnit.MILLISECONDS);

```

上面这块代码，其实是启动了一个定时调度的任务，他默认是每隔30s就会执行一次Broker注册的过程，上面的registerNameServerPeriod是一个配置，他默认的值就是30s一次。

所以其实大家看到这里就会明白，默认情况下，第一次发送注册请求就是在进行注册，就是我们上一讲讲的内容，他会把Broker路由数据放入到NameServer的RouteInfoManager的路由数据表里去。

但是后续每隔30s他都会发送一次注册请求，这些后续定时发送的注册请求，其实本质上就是Broker发送心跳给NameServer了，我们看下图示意。



那么后续每隔30s，Broker就发送一次注册请求，作为心跳来发送给NameServer的时候，NameServer对后续重复发送过来的注册请求（也就是心跳），是如何进行处理的呢?

说到这里，我今天来带大家看一下RouteInfoManager的注册方法逻辑。

上一次是给大家留了作业，想必很多人可能都已经看出一点感悟来了，今天就我们一起来分析一下，下面是RouteInfoManager的注册Broker的源码。

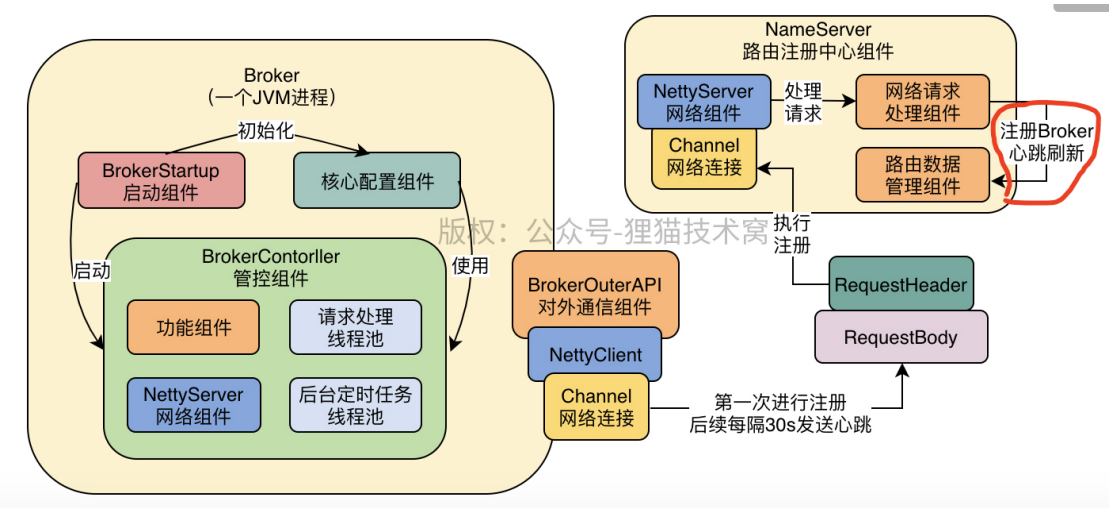
```
public RegisterBrokerResult registerBroker(
  final String clusterName,
  final String brokerAddr,
  final String brokerName,
  final long brokerId,
  final String haServerAddr,
  final TopicConfigSerializeWrapper topicConfigWrapper,
  final List<String> filterServerList,
  final Channel channel) {
  RegisterBrokerResult result = new RegisterBrokerResult();
  try {
     try {
         // 这里是加写锁，同一时间，只能是一个线程来执行
        this.lock.writeLock().lockInterruptibly();

         // 下面这里是根据clusterName获取了一个set集合
         Set<String> brokerNames = this.clusterAddrTable.get(clusterName);
         if (null == brokerNames) {
             brokerNames = new HashSet<String>();
             this.clusterAddrTable.put(clusterName, brokerNames);
         }
           // 然后直接就把brokerName扔到了这个set集合里去
           // 这就是在维护一个集群里有哪些broker存在的一个set数据结构
           // 假如你后续每隔30s发送注册请求作为心跳，这里是没影响的
           // 因为同样一个brokerName反复发送，这里set集合是自动去重的!
           brokerNames.add(brokerName);

           boolean registerFirst = false;

```

```
            // 这里是根据brokerName获取到BrokerData
            // 他用一个brokerAddrTable作为核心路由数据表
            // 这里存放了所有的Broker的详细的路由数据
           BrokerData brokerData = this.brokerAddrTable.get(brokerName);

            // 如果第一次发送注册请求，这里就是null
            // 那么就会封装一个BrokerData，放入到这个路由数据表里去
            // 其实这个就是核心的Broker注册过程

            // 如果后续每隔30s发送注册请求作为心跳，这里是没影响的
            // 因为明显你重复发送注册请求的时候，这个BrokerData已经存在了
            // 他不会重复进行处理的
           if (null == brokerData) {
               registerFirst = true;
               brokerData = new BrokerData(clusterName, brokerName, new HashMap<Long, String>());
               this.brokerAddrTable.put(brokerName, brokerData);
           }

              // 下面这里是对路由数据做一些处理，暂时不用管
              Map<Long, String> brokerAddrsMap = brokerData.getBrokerAddrs();
              //Switch slave to master: first remove <1, IP:PORT> in namesrv, then add <0, IP:PORT>
              //The same IP:PORT must only have one record in brokerAddrTable
              Iterator<Entry<Long, String>> it = brokerAddrsMap.entrySet().iterator();
              while (it.hasNext()) {
                Entry<Long, String> item = it.next();
                if (null != brokerAddr && brokerAddr.equals(item.getValue()) && brokerId != item.getKey()) {
                    it.remove();
                }
            }

            String oldAddr = brokerData.getBrokerAddrs().put(brokerId, brokerAddr);
            registerFirst = registerFirst || (null == oldAddr);

```

```
            if (null != topicConfigWrapper
                && MixAll.MASTER_ID == brokerId) {
                if (this.isBrokerTopicConfigChanged(brokerAddr, topicConfigWrapper.getDataVersion())
                    || registerFirst) {
                    ConcurrentMap<String, TopicConfig> tcTable =
                      topicConfigWrapper.getTopicConfigTable();
                    if (tcTable != null) {
                        for (Map.Entry<String, TopicConfig> entry : tcTable.entrySet()) {
                           this.createAndUpdateQueueData(brokerName, entry.getValue());
                        }
                    }
                }
            }

              // 其实比较核心的在这里，这就是每隔30s发送注册请求作为心跳的时候
              // 他最核心的处理逻辑
              // 说白了，他就是每隔30s都会封装一个新的BrokerLiveInfo放入Map
              // 所以每隔30s，最新的BrokerLiveInfo都会覆盖之前上一次的BrokerLiveInfo
              // 这个BrokerLiveInfo里，就有一个当前时间戳，代表你最近一次心跳的时间
              // 这就是Broker每隔30s发送注册请求作为心跳的处理逻辑
             BrokerLiveInfo prevBrokerLiveInfo = this.brokerLiveTable.put(brokerAddr,
               new BrokerLiveInfo(
                 System.currentTimeMillis(),
                 topicConfigWrapper.getDataVersion(),
                 channel,
                 haServerAddr));
             if (null == prevBrokerLiveInfo) {
                 log.info("new broker registered, {} HAServer: {}", brokerAddr, haServerAddr);
            }
```

```
                // 下面的代码暂时先不用管了
                if (filterServerList != null) {
                    if (filterServerList.isEmpty()) {
                        this.filterServerTable.remove(brokerAddr);
                    } else {
                        this.filterServerTable.put(brokerAddr, filterServerList);
                    }
                }

                if (MixAll.MASTER_ID != brokerId) {
                    String masterAddr = brokerData.getBrokerAddrs().get(MixAll.MASTER_ID);
                    if (masterAddr != null) {
                        BrokerLiveInfo brokerLiveInfo = this.brokerLiveTable.get(masterAddr);
                        if (brokerLiveInfo != null) {
                            result.setHaServerAddr(brokerLiveInfo.getHaServerAddr());
                            result.setMasterAddr(masterAddr);
                        }
                    }
                }
            } finally {
                this.lock.writeLock().unlock();
            }
        } catch (Exception e) {
            log.error("registerBroker Exception", e);
        }

        return result;
    }

```

所以我们看下图，有一个红色圈圈示意了，每隔30s你发送注册请求作为心跳的时候，RouteInfoManager里会进行心跳时间刷新的处理。



接着我们来思考最后一个问题，那么假设Broker已经挂了，或者故障了，隔了很久都没有发送那个每隔30s一次的注册请求作为心跳，那么此时NameServer是如何感知到这个Broker已经挂掉的呢？

我们重新回到NamesrvController的initialize()方法里去，里面有一个代码是启动了RouteInfoManager中的一个定时扫描不活跃Broker的线程。

```
this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {

    @Override
    public void run() {
      NamesrvController.this.routeInfoManager.scanNotActiveBroker();
    }

}, 5, 10, TimeUnit.SECONDS);

```

上面这段代码，就是启动一个定时调度线程，每隔10s扫描一次目前不活跃的Broker，使用的是RouteInfoManager中的scanNotActiveBroke()方法，我们去看看那个方法的逻辑，就知道他如何感知到一个Broker挂掉了。

```
public void scanNotActiveBroker() {
  // 大家可以看到，他这里就是扫描BrokerLiveInfo这个心跳数据结构
  // 他里面遍历一下，就可以拿到每个Broker最近一次心跳刷新的BrokerLiveInfo
  // 也就知道一个Broker最近一次发送心跳是什么时候
  Iterator<Entry<String, BrokerLiveInfo>> it = this.brokerLiveTable.entrySet().iterator();
  while (it.hasNext()) {
    Entry<String, BrokerLiveInfo> next = it.next();
    long last = next.getValue().getLastUpdateTimestamp();
     // 下面这就是核心判断逻辑
     // 如果说当前时间距离上一次心跳时间，超过了broker心跳超时时间，默认是120s
     // 也就是说，如果一个Broker两分钟没发送心跳，就认为他已经死掉了
    if ((last + BROKER_CHANNEL_EXPIRED_TIME) < System.currentTimeMillis()) {
        RemotingUtil.closeChannel(next.getValue().getChannel());
        it.remove();
        log.warn("The broker channel expired, {} {}ms", next.getKey(), BROKER_CHANNEL_EXPIRED_TIME);
         // 此时在这里就会把这个Broker从路由数据表里都剔除出去
        this.onChannelDestroy(next.getKey(), next.getValue().getChannel());
    }
   }
}
```

今天给大家留一个源码分析的小作业，就是把Broker的注册、□心跳以及故障发现的相关源码都看一遍，同时结合我们画的图，深刻的理解和记忆Broker跟NameServer的交互流程，核心组件。

End

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情留言提问，我会逐一答疑

如何加群：购买狸猫技术窝专栏的小伙伴都可以加入狸猫技术交流群，一个非常纯粹的技术交流的地方

具体加群方式，请参见目录菜单下的文档：《付费用户如何加群》（**购买后可见**）